XML REMOTE PROCEDURE CALL (XML-RPC)

The present application is based on and claims priority from Application No. 60/079,100 filed March 23, 1998 and Application No. 60/096,909 filed August 17, 1998, the disclosures of which are incorporated by reference herein.

Background of the Invention

Remote Procedure Call (RPC) is a mechanism by which an application residing on one machine requests the services of an application residing on another machine. A machine (as the term is used here and in the appended claims) may be a physical CPU or computer, a virtual CPU or virtual computer (otherwise known as a virtual machine or VM), or it may be a space of addressable memory in a system that isolates applications or processes so that they may not directly invoke one another. RPC requires one application to send one or more messages to another application in order to invoke a procedure of the other application. The recipient application may reply by sending one or more messages back to the requesting application. The term "RPC" identifies both the mechanism by which this occurs and the instance of such an exchange between applications.

Applications engaged in RPC conventionally do not message each other directly. Instead they rely on intervening software called "middleware" to translate between the wire protocol and the programming-language-specific functions or class methods of the application. This application uses the term "service" to generically refer to a function (including a procedure or a class method) or any action to be taken or performed, including access to web sites and databases. The term "wire protocol" refers to the transfer-protocol/message-encoding combination that the programs use as their common language for communication. The transfer-protocol is the mechanism by which the message transfers between the applications, and the message-encoding is the format in which the message is represented. The Object Management Group's IIOP (OMG Internet Inter-ORB Protocol) and the Open Software Foundation's DCE (OSF Distributed Computing Environment) are examples of wire protocols, and each defines both a transfer-protocol and a message-encoding. RPC has in the past been implemented using IIOP, DCE, or proprietary protocols. HTTP is an example of a transfer-protocol that is independent of message-encoding. The IIOP and DCE protocols have themselves been encoded into messages that tunnel through HTTP.

Each message-encoding is a particular format for representing a message. Any set of data may be represented in a particular format, not just RPC messages. This application uses the term 'data-package' to refer to each such set of data, and it uses the term 'encoding' to refer to the format in which a data-package is represented. An RPC message is a kind of data-package, and a message-encoding is the encoding of an RPC message. A tab-delimited file is an example of a simple encoding. The binary representation of a C programming language data structure is an example of a more complex encoding. A data-package may be thought of as a collection of individual data items. In a tab-delimited file, each row is a data item, as is each field in each row. In a C data structure, each field is a data item, as is each nested data structure and each field in each nested data structure. Notice that a data item may contain one or more other data items.

An 'argument' is a particular kind of data item. The term argument is used in the software industry to refer to a data item that is passed to a function or that is returned from a function. When the signature of a function is represented in the Java programming language, the arguments of the function consist of the comma-delimited terms that appear in parentheses and the term that represents the return value of the function. Consider the following function signature:

```
EmployeeID addEmployee(Name n, Address a, JobType t)
```

This function has four arguments whose classes are EmployeeID, Name, Address, and JobType. The Address class might be defined as follows:

```
class Address {
    String street;
    String city;
```

```
        String state;
        String zip;
    }
```

In other contexts one might interpret the street, city, state, and zip data items to each be an argument of the function, but this application defines 'argument' such that these data items are not arguments. The arguments of a function are those data items that the signature of the function identifies, where for purposes of this definition the definitions of the argument types are not considered to be part of the signature.

A function is only one kind of service, and XML RPC messages can exist in the absence of functions. Hence, it is necessary to define the term 'argument' in terms of the message encoding. Therefore, given a data item X contained in message M1 expressed in encoding E, X is an argument when it satisfies both of the following requirements: (1) there exists a message M2 also expressed in E such that M2 is constructed by adding a data item Y to M1 where neither X contains Y nor Y contains X; and (2) X is not contained in any data item having property (1). It is possible to define an encoding that hard-codes arguments relative to another encoding so that the defined encoding expresses a subset of the messages of the other encoding. Consequently, a message cannot be examined for arguments without interpreting the message with respect to a particular encoding.

Each argument is an input argument, an output argument, or both an input and an output argument. The return value is always an output argument, but function signatures usually do not indicate which if the remaining arguments are input arguments or output arguments. The input arguments and output arguments must be established by convention. The input arguments are those arguments that the software that invokes the function uses to pass data items to the function. The output arguments are those arguments that the function uses to pass data items to the software that invoked the function. It is possible for one argument to be used for both purposes. In this case, the argument appears in both the RPC request message and the RPC reply message, although the values of the occurrences may differ. The OMG CORBA specification provides further treatment of the distinction between input and output arguments in the context of RPC.

In addition to containing data items, a data-package may contain labels. A label is an identifier that provides information about one or more data items. A label that provides information about a data item is said to be a label for the data item. No data item is a label, since the encoding assumes responsibility for providing the association between the data item and the label.

This application defines two kinds of labels: type labels and semantic labels. A type label identifies the data type of each associated data item. Data types include the primitive data types that programming languages define. The primitive data types include integers, floats, longs, strings, and booleans. According to this definition, data types also include primitive data structures, such as arrays, records, and vectors (e.g. the Java Vector class). Data types provide the information that is necessary to represent the data item in memory so that a programming language may use the data item. However, there is no requirement that any two programming languages or any two implementations of RPC represent the same data types in the same way.

A semantic label is an identifier that names an associated data item according to the data item's role within the message. The purpose is to allow software that consumes the message to locate data items by role. The semantic label of an argument is analogous to the name that a method signature gives to the argument. Again consider the following signature:

```
EmployeeID addEmployee(Name n, Address a, JobType t)
```

This function has arguments named 'n', 'a', and 't'. It also has an unnamed argument whose type is EmployeeID. The names may be used as semantic labels for their associated arguments. The semantic label of a field found in a class, record, or structure is analogous to the name given to that field. Consider the previously given definition of the Address class. It has four data items. Each has a data type of string, and their names are 'street', 'city', 'state', and 'zip'. These names may be used as the semantic labels for the associated data items. However, unlike the names of a programming language structure, multiple data items may have the same semantic labels. For example, two data items may each have the 'street' label to indicate that they each represent one line of the street address. On the other hand, an array or a vector may contain multiple data items such that a semantic label is associated with the entire array or vector but not with each data item.

This application classifies encodings as either self-describing or non-self-describing. A self-describing encoding associates a semantic label with each argument and includes both the label and the association within each data-package. A completely self-describing encoding associates a semantic label with every data item except possibly for the data item constituents of arrays and vectors. A semantic label allows a program to identify a data item without requiring the program to know in advance the location of the data item within the data-package and without requiring the program to know the exact structure of the data-package. One example of a data-package that uses a self-describing encoding is a tab-delimited file that begins with a row consisting of the names of each field in a row. Another self-describing encoding is the binary representation of a hash table, which assigns a name (or key) to each data item in the table. Non-self-describing encodings do not contain semantic labels for their arguments. A program that reads a data-package expressed in a non-self-describing encoding must assign semantics to each argument by applying information found in the application and not in the data-package. For example, a program that reads a tab-delimited file to load records into a database must know in advance how to correlate the fields in the file to the columns of the database table.

This application also classifies encodings as either specific or generic. A specific encoding is an encoding that is designed to represent an application-specific or industry-specific data structure. A generic encoding is an encoding that is designed to represent any or nearly any data structure, regardless of application or industry. For example, the ASCII form of a database report is a specific encoding since it can express the data found in a particular database but not the data found in any database. Packet headers in communications protocols also use specific encodings. An implementation of the C programming language uses a generic encoding to represent in-memory data structures; this is a generic encoding because one encoding scheme is capable of representing any C data structure.

Finally, this application distinguishes between binary encodings and text-based encodings. A text-based encoding is a human-readable and human-writable encoding. Data-packages expressed in a text-based encoding can be loaded into a text editor and can be read and modified by humans using the text editor. A human can also create a data-package expressed in a text-based encoding by writing the message completely from scratch using the text editor. Text editors include vi, emacs, MS-DOS EDIT.EXE and Microsoft Notepad. Binary encodings are those encodings that do not satisfy these human-accessibility criteria. Tab-delimited files are examples of data expressed in a text-based encoding, while the in-memory representation of C data structures is an example of a binary encoding.

The IIOP and DCE wire protocols use non-self-describing generic binary encodings. Of all the example encodings discussed in this application, the IIOP and DCE encodings are most closely analogous to the in-memory binary representations of C or C++ data structures. However, the analogy is not perfect because the encodings have been designed to support the data structures found in a variety of languages, and because they have been designed to support the translation of data between a variety of platforms.

Although IIOP and DCE are non-self-describing, it is possible to define services in IIOP and DCE that accept self-describing parameters and that invoke other services using the identified parameters. Microsoft's COM (Component Object Model) accomplishes this on top of DCE through the set of services found in its IDispatch interface. By invoking the services of IDispatch, an application can emulate the functionality that would be available through a wire protocol that is natively self-describing. However, DCE remains non-self-describing, since the services of IDispatch are expressed in the same non-self-describing encoding in which all other service invocations of the protocol are expressed. IDispatch effectively defines a high level protocol that is layered on top of DCE. More work is required to develop software that invokes self-describing services through IDispatch than is required to develop software that invokes the native non-self-describing services of DCE. Additional effort is required because to invoke a single self-describing service the developer must write software that conforms to the IDispatch protocol. Invoking a single non-self-describing service only requires the developer to invoke a single function in the code that DCE IDL compilers automatically generate for the developer.

IIOP defines a data-type called 'ANY'. A data item expressed in this type is a container for other data items. The ANY data type associates a type label with each data item that it contains, but it does not associate a semantic label with any of these data items. It also does not associate a type label with the instance of the ANY data item itself, unless the instance happens to be contained within another ANY data item. Consequently, an implementation of IIOP that receives a message containing an instance of the ANY data type cannot determine that the instance is of type ANY simply by examining the message. The recipient of the IIOP message must know in advance that the argument is of type ANY. In fact, the recipient must know the types of all the arguments in order to extract the arguments from the message. This aspect of IIOP makes programs that use IIOP strongly sensitive to changes in the messages. Adding, removing, or changing argument types in the programs that generate the IIOP messages necessitates making analogous changes in all the programs that receive the messages. This would not be necessary in programs that relied on data type labels that the message itself provides.

IIOP and DCE are also stateful, bi-directional, and complex. These protocols are stateful because the client and server must maintain state information to successfully transfer messages between them. The protocols are bi-directional because the server must sometimes asynchronously send messages to the client. Finally, IIOP and DCE are complex because they are designed to provide the union of the features found in a wide variety of programming languages. This latter property of IIOP and DCE allows any two programs to communicate with service invocations that support the rich variety of invocation mechanisms that are common to the programming languages in which the two programs are written.

Traditional middleware implementations are based on the IIOP and DCE protocols and consequently suffer from the following drawbacks:

- Because the encodings are generic, translation software is required to allow RPC messaging between a program that uses a specific encoding and a program that uses a generic encoding. Translation software is also required between programs that use different specific encodings. If IIOP or DCE is to be the wire protocol, the translation software must be installed between each program that uses a specific encoding and the communications channel that connects the programs. Furthermore, generic encodings are necessarily more removed from the data structures that are common to a given industry, which prevents most industry experts from examining messages, since expertise with the generic encoding would be required in addition to expertise with the industry data structures.

- Because the encodings are binary, special software is required to read, modify, or write the RPC messages. If the message is expressed in a non-self-describing encoding, the special software must be application-specific or industry-specific software. These factors limit the accessibility of messages expressed in the IIOP and DCE encodings, since the translation software must exist, and since one must have translation software on hand to access the messages.

- Because these encodings are non-self-describing, programs can't extract and selectively process the data items of an arbitrary message without first being specifically configured to recognize the particular kind of message. Programs that do support configured access to non-self-describing messages generally require separate configuration information for each kind of non-self-describing message. As the message-generating programs evolve the messages and add new message types, the configuration information of the other programs must be updated. To further complicate matters, each program may require different configuration details or separate configuration entry procedures. Finally, if a message-generating program changes the format of a non-self-describing message, the message-consuming programs must also change their knowledge of that message, since otherwise the these programs would not be able to extract and identify the data items found in the message.

- Because the IIOP and DCE protocols are stateful, the protocols are not well-suited for messaging using HTTP, since HTTP is a stateless protocol. HTTP provides only a limited mechanism for maintaining client state. Under this mechanism, the client maintains the state information, which means that client-provided state information cannot be trusted to belong to the same client over time, unless the client connects through an authenticated connection. One must therefore inventively design a stateful protocol on top of HTTP in order to tunnel IIOP or DCE through HTTP. It is desirable to use HTTP as the transfer protocol because of the ubiquity with which it is installed on operating systems, and because it is typically the only protocol that network firewalls allow to pass.

- Because these protocols are bi-directional, the protocols are not well-suited for messaging using HTTP, since HTTP is unidirectional. For example, an IIOP client may register a callback function with an IIOP server. When it comes time for the server to invoke the callback function, the server cannot issue an HTTP request to the client unless the client is also configured to be an HTTP server. Administrative and security requirements normally make it undesirable to configure a client as an HTTP server. Therefore, one must inventively design a bi-directional protocol on top of HTTP in order to support IIOP and DCE bi-directional behavior.

- Because the encodings are intended to bridge applications written in almost any two programming languages on almost any two platforms, the encodings are proportionately complex. IIOP and DCE take the superset approach to bridging applications by ensuring that the encoding can represent nearly every kind of service invocation that programming languages are capable of expressing. To create software that uses IIOP or DCE, a software developer must define the signatures of these services. The superset approach complicates the definitions and minimizes the pool of skilled experts that can develop for IIOP and DCE.

This approach also complicates the tools that manipulate messages expressed in these encodings.

- Because of the complexity of the IIOP and DCE protocols, implementations from two different vendors are generally incompatible. To get two programs to communicate over a network, the computers running the programs must conventionally run middleware software from the same vendor. Vendors have implemented gateway software to bridge between different implementations of these protocols. The CORBA 2.0 specification has mitigated this problem to a large degree, but it has not eliminated it completely. Thus, integrating programs residing in different organizations often requires either both organizations to agree on the installation or one company to buy additional software to provide a gateway between the existing middleware installations.

Therefore, there is a need for improved RPC techniques that do not suffer from the above drawbacks. The techniques would provide the following benefits:

- It would be beneficial to have an RPC mechanism that allows messages to be expressed in application-specific or industry-specific encodings. This would allow programs to communicate without requiring messages expressed in specific encodings to be translated into messages expressed in generic encodings. Provided that the messages were also text-based, this would also allow industry experts to create and examine messages without requiring that they have expertise in a generic encoding. Text-based messages in specific encodings would also be easier to read than text-based messages in generic encodings, since the generality of generic encodings makes generically-encoded messages more verbose.
- It would be beneficial to have an RPC mechanism that uses text-based messages, since special software would not be needed to read, write, or modify the messages. Text editors would suffice. People who have expertise in industry data structures would have less reliance on the expertise of others to translate messages into a form that they can understand.